Querying Complex Graphs[⋆]

Yanhong A. Liu and Scott D. Stoller

Computer Science Department, State University of New York at Stony Brook
Stony Brook, NY 11794
{liu,stoller}@cs.sunysb.edu

Abstract. This paper presents a powerful language for querying complex graphs and a method for generating efficient implementations that can answer queries with complexity guarantees. The graphs may have edge labels that may have parameters, and easily and naturally capture complex interrelated objects in object-oriented systems and XML data. The language is built on extended regular path expressions with variables and scoping, and can express queries more easily and clearly than previous query languages. The method for implementation first transforms queries into Datalog with limited extensions. It then extends a previous method to generate specialized algorithms and complexity formulas from Datalog with these extensions.

1 Introduction

Database applications must query complex interrelated objects, and thus languages that provide both the power and ease of querying complex graphs are highly desired. Such query languages are essential not only for traditional database applications and mining of semi-structured data, but also for analyzing large computer programs and systems.

Various forms of regular path queries are ways of declaratively expressing queries on graphs as regular-expression-like patterns that are matched against paths in the graph. Some have been used widely in querying semi-structured data (e.g., [1, 3, 7, 20]), including in particular tree structured data in XML, which is increasingly used for representing data, including knowledge as data and programs as data. Some more powerful kinds have provided general frameworks for analyzing computer programs and systems (e.g., [21, 11, 17]).

Regular-expression-like patterns are composed of simple and easy operations for sequencing, choice, repetition, skipping, negation, etc. Even though they are not as powerful as languages in more sophisticated frameworks, they are more perspicuous and convenient, and are sufficiently powerful to express common and important properties. The combined power and simplicity contribute to their wide use in computing, in database and web information retrieval, languages and compilers, operating systems and security, etc.

⋆ This work was supported in part by ONR under grants N00014-04-1-0722 and N00014-02-1-0363 and NSF under grants CCR-0306399 and CCR-0311512.

While regular-expression-like patterns have been studied and used extensively in analysis of linear data and in recent years tree-structured data, many applications deal with much more complex interrelated objects. In regular path query frameworks, such information is captured as graphs, and the analyses are based on properties that hold on paths in the graph. In particular, parametric regular path queries [17] allow the use of variables, also called parameters, in queries so that additional information along paths can be captured and related.

Despite this progress, no previous regular path query framework supports easy, powerful, and efficient queries over a rich data model that naturally models all aspects of objects in object-oriented systems and XML data. Frameworks with rich data models exist for querying object-oriented databases [14] and for querying XML data [20], but the former does not support regular-expression like patterns, and the latter does not support queries on graphs. Languages that support regular-expression like patterns on graphs [10, 13] are studied heavily in terms of expressiveness and query containment, but not on improved ease of expressing queries or efficient implementation with precise complexity guarantees. The best of existing approaches must be combined and extended to support easy, powerful, and efficient queries of complex graphs.

This paper presents a powerful language for querying complex graphs and a method for generating efficient implementations that can answer queries with complexity guarantees. The graphs may have edge labels that may have parameters and easily and naturally capture complex interrelated objects in object-oriented systems and XML data. The language is built on parameterized regular path expressions with copings, and can express queries more easily and clearly than previous query languages. The method for implementation first transforms queries into Datalog with limited extensions. It then extends a previous method [18] to generate specialized algorithms and complexity formulas from Datalog with these extensions.

2 The data model

Complex graphs. We consider edge-labeled directed graphs where the labels may have parameters. We call such graphs *complex graphs*. A complex graph comprises a set of vertices and a set of edges. Each vertex has a unique id. Each edge has a source vertex, a target vertex, and a label.

A label captures information relating a source vertex and a target vertex. For example, in applications that manipulate computer programs, an edge may relate a program-point vertex to another program-point vertex with a label `def` that captures the assignment operation in between. In a supply chain application, an edge may relate a manufacturer vertex and a product vertex with a label `supply`. A label may have arguments that capture additional information about the relationship. For example, an assignment operation `num` := `5` may be represented using an edge label `def`(`num`) or `def`(`num`, `5`). To represent the date and means of a supply relationship, a label such as `supply`(`12/20/04`, `air`) may be used. A special label can be used to indicate that no information about the relationship is of interest.

We refer to names, such as `def` and `supply`, that represent kinds of relationships, as *constructors*. We refer to names, such as `num`, `12/20/04`, and vertex ids, that represent individuals, as *constants*. A *label* is a constructor applied to zero or more arguments, where each argument is a constant. We assume that the domains of constructors and constants are finite; this assumption always holds in any particular application.

Modeling objects and relationships. Complex graphs can model objects and classes naturally and precisely. Objects are modeled as vertices, where vertex ids are object ids. Values of attributes are also modeled as vertices, consistent with them being objects in a pure object-oriented model. Classes are also modeled as vertices, consistent with classes being objects in a powerful object-oriented model.

Attributes and relationships are modeled as edges. An edge labeled with an attribute name connects an object to the value of that attribute of the object. An *instance-of* relationship connects an object to another object that is an instance of the first object. A *subclass* relationship connects an object to another object that represents a subclass of the first object.

Modeling XML data. Complex graphs can model XML data easily and significantly better than using only trees. XML elements and attribute values are modeled as vertices. XML nested element relationship and attribute are modeled as edges relating an element to a child element of it and to the value of an attribute of it, respectively; these are the straightforward tree edges in XML documents. Relationships that can not be captured using tree edges are expressed directly as graph edges in our model but need to be encoded using `IDREF` and `IDREFS` in XML.

Paths. A *path* in a complex graph is expressed as a sequence of vertices and edges of the form:

$$[\mathtt{v_0}]\,\mathtt{l_1}\,[\mathtt{v_1}]\,\mathtt{l_2}\,\ldots\,[\mathtt{v_{n-1}}]\,\mathtt{l_n}\,[\mathtt{v_n}] \tag{1}$$

where each $\mathtt{v_i}$ is a vertex, and each $\mathtt{l_i}$ is the label of an edge from $\mathtt{v_{i-1}}$ to $\mathtt{v_i}$. In another word, the above expression asserts that there is an edge labeled $\mathtt{l_i}$ from vertex $\mathtt{v_{i-1}}$ to $\mathtt{v_i}$. For example, the program point `start` followed by an operation `prompt` followed by the program point `prelogin` followed by an operation `read(account, password)` followed by the program point `preauthentication` may be represented as:

$$[\mathtt{start}]\,\mathtt{prompt}\,[\mathtt{prelogin}]\,\mathtt{read(account,password)}\,[\mathtt{preauthentication}]$$

3 Path-based queries

For ease of presentation, in this section, we use `x`, `y`, and `z` possibly with subscripts for variables, and use other names besides keywords for constructors and constants.

Simple queries. One may query for vertices, labels, constructors, and arguments that satisfy certain properties based on paths. Simple queries are of the form:

$$\mathtt{x_1, ..., x_k :\ e} \tag{2}$$

where $\mathtt{x_1, ..., x_k}$ are variables, called *query variables*, and $\mathtt{e}$ is an expression, called a *path-properties expression*, and is constructed from paths that contain $\mathtt{x_1, ..., x_k}$ and may contain other variables, wildcard $\_$, and negation $\neg$; from combinations of paths using conjunction $\wedge$, disjunction $\vee$, and negation $\neg$; and from constraints added to these that involve primitive arithmetic, comparison, and Boolean operations on variables in the paths. The query returns the set of tuples of values of $\mathtt{x_1, ..., x_k}$ such that there exist values of the other variables, if any, for which the properties about paths asserted by the expression $\mathtt{e}$ hold.

Variables and wildcard may refer to, and negation may be applied to, vertices, labels, constructors, and arguments. Variables that refer to labels may not refer to vertices, constructors, or arguments. Multiple occurrences of a variable must be bound to the same value. A wildcard matches any value. A negation applied to an item matches any value other than what the item matches. For example, the following query returns the set containing each object that is a branch of `acme` and has a director whose salary is at least `150000`:

$$\mathtt{x :\ [acme]\,branch\,[x]\,director\,[\_]\,salary\,[y]\ \wedge\ y >= 150000}$$

We could easily query also the salary, by returning $\mathtt{x, y}$. Each variable used in a constraint must also appear outside of the constraint, like $\mathtt{y}$ appears in $\mathtt{salary\ [y]}$. Note that path-properties expression $\mathtt{[v_0]\,l_1\,[v_1]\,l_2\,[v_2]}$ is equivalent to expression $\mathtt{[v_0]\,l_1\,[v_1]\ \wedge\ [v_1]\,l_2\,[v_2]}$.

Extended regular expression based queries. One may also express the property that a path is formed by repeating a path segment 0 or more times. This is done by applying the repetition operator * to the repeated segment. For example, the following query returns the set of program points $\mathtt{y}$ that immediately follow a use of an uninitialized variable, i.e., there is a path from program point `start` on which a variable is not defined and is used right before $\mathtt{y}$:

$$\mathtt{y :\ [start]\,(\neg def(x)[\_])^*\,use(x)\,[y]} \tag{3}$$

One may return also the uninitialized variable by including $\mathtt{x}$ as another query variable.

Often, intermediate vertices in paths are not of interest, as in the example above. Thus we allow $[\_]$ to be omitted from a path; note that this also allows us to easily refer to the program point right before $\mathtt{use(x)}$ without unrolling the last iteration of the repetition. We also allow a shorthand $|$, instead of using $\vee$, to separate alternative paths. Queries that may use these notations are called *extended regular expression based queries*. For example, the following query returns the set of program point pairs $\mathtt{z, y}$ right before and after, respectively, the *first* use of an uninitialized variable:

$$\mathtt{z, y :\ [start]\,(\neg(def(x)|use(x)))^*\,[z]\,use(x)\,[y]} \tag{4}$$

Extended regular expression based queries provide the full power and ease of using extended regular expressions in queries over parameterized edge labels, as in *parametric regular path queries* [17]. Parametric regular path queries do not support the use of vertex ids as the queries in this paper do. This support allows us to easily query vertices on cycles, i.e., a vertex is returned if some nonempty path from it goes back to it:

$$\mathtt{x} : [\mathtt{x}]\ \_^{+}\ [\mathtt{x}]$$

where $\mathtt{s}^{+}$ is a short hand for $\mathtt{s}\,\mathtt{s}^{*}$.

Variable scoping and nested queries. Variables can be declared with a scope local to a subexpression. That is, a path-properties expression may be of the form:

$$\mathtt{local}\ \mathtt{x}_1, ..., \mathtt{x}_\mathtt{k}\ \mathtt{e} \tag{5}$$

where the keyword `local` indicates that the scope of variables $\mathtt{x}_1, ..., \mathtt{x}_\mathtt{k}$ is $\mathtt{e}$. For example, in a model of a computer network, where `link` relates directly connected nodes, the following query returns all pairs of a client and a server such that the two are connected by a path containing nodes that do not block port 22:

$$\begin{aligned} \mathtt{x}, \mathtt{y} : {} & [\mathtt{x}]\,\mathtt{type}\,[\mathtt{client}] \wedge [\mathtt{y}]\,\mathtt{type}\,[\mathtt{server}] \wedge \\ & [\mathtt{x}]\ (\mathtt{local}\ \mathtt{z}\ \mathtt{link}\,[\mathtt{z}] \wedge \neg([\mathtt{z}]\,\mathtt{block}\,[22]))^{*}\ \mathtt{link}\,[\mathtt{y}] \end{aligned}$$

Note that when scoping is not inside a repetition, it is unnecessary and can be removed, by replacing each local variable with a fresh variable. Variables declared inside a repetition can not be replaced this way because such a variable is local to the repeated expression and may be bound to different values for different rounds in the repetition, but a non-local variable must be bound to the same value for all rounds of the repetition.

A query may also be nested inside [ ] to express the properties of the vertex in it, in the form of $[\mathtt{x} : \mathtt{e}]$, and it is equivalent to conjuncting the expression $\mathtt{e}$ to the immediately enclosing expression. For example, the segment repeated in the example above can also be written as

$$\mathtt{local}\ \mathtt{z}\ \mathtt{link}\,[\mathtt{z} : \neg([\mathtt{z}]\,\mathtt{block}\,[22])]$$

This adds convenience and modularity using only the concepts and syntax already introduced.

Querying objects and XML data. Objects can be organized into classes, and methods can be defined in classes as usual for querying objects [14], except that extended regular expression based queries can now be used in the method body. The use of extended regular expressions is essential for querying graph structures of unbounded size, and it greatly increases the expressive power of the query language.

When a method `m` returns exactly one query variable, an invocation of `m` can have the same syntax and semantics as a short-cut edge, $[\mathtt{v}_1]\mathtt{m}(\mathtt{a}_1, ..., \mathtt{a}_\mathtt{k})[\mathtt{v}_2]$, where the starting vertex $\mathtt{v}_1$ is an object on which `m` is invoked, $\mathtt{a}_1, ..., \mathtt{a}_k$ are other arguments to `m`, and the ending vertex $\mathtt{v}_2$ is an object returned by `m`. For simplicity, we may use the same name space for edge label constructors and method names, and may allow the same names to be used for both and give preference to one of them.

Objects can be created out of the end result of a query as usual, by viewing each returned tuple as an object, giving it a logical object id, and giving an attribute name to each component of the tuple [14]. Since these objects are created after query evaluation, all operations including repetitions in a query operate on finite data, and therefore we can guarantee that all queries terminate.

Querying XML data is easy using graph queries. Unbounded levels of element nesting poses a challenge to previous object query languages [14] but is easily expressed in our language using the repetition operator. Querying complicated graphs using our language is significantly easier than using XML query languages, such as XQuery, that employ explicit joins for relationships that are not nested elements or attributes.

Expressiveness. We think this query language has the same expressiveness as GraphLog [10], which is equivalent to stratified linear Datalog, first order logic with transitive closure, and non-deterministic logarithmic space. This is because our language supports all the kinds of graph edges and query operations that GraphLog does, and variable scoping and query nesting in our language can be translated into GraphLog.

Support for scoping, and textual flexibilities such as query nesting, make our language easier to use, either by itself or as part of another query language such as [15]. For example, the client-server example above, if expressed using GraphLog, needs two graphs, one for each of the following rules:

```
result(x,y) :- type(x,client), type(y,server),
               link_node_not_block_22*.link(x,y).
link_node_not_block_22(x,z) :- link(x,z), not block(z,22).
```

where each argument variable or constant corresponds to a vertex, and each `label(vert1, vert2)` corresponds to an edge from `vert1` to `vert2` and labeled `label`; the edge on the left of `:-` is called the distinguished edge of the graph, and is drawn as a thick line. So the first graph has 4 vertices and 4 edges, and the second has 3 vertices and 3 edges. Furthermore, if there are additional constraints involving `z`, `x`, and `y`, one can simply conjunct them with the segment repeated in our language, but one must add not only these constraints to the second rules, but also additional parameters to the label `link_node_not_block_22` in both graphs to pass them between the graphs.

4 Transformation into Datalog with limited extensions

Datalog with limited extensions. A Datalog program is a finite set of relational rules of the form:

$$\mathtt{p_1}(\mathtt{x_{11}}, ..., \mathtt{x_{1a_1}}) \wedge ... \wedge \mathtt{p_h}(\mathtt{x_{h1}}, ..., \mathtt{x_{ha_h}}) \rightarrow \mathtt{q}(\mathtt{x_1}, ..., \mathtt{x_a}) \tag{6}$$

where $\mathtt{h}$ is a natural number, each $\mathtt{p_i}$ (respectively $\mathtt{q}$) is a relation of $\mathtt{a_i}$ (respectively $\mathtt{a}$) arguments, each $\mathtt{x_{ij}}$ and $\mathtt{x_k}$ is either a constant or a variable, and variables in $\mathtt{x_k}$'s must be a subset of the variables in $\mathtt{x_{ij}}$'s. If $\mathtt{h} = 0$, then there are no $\mathtt{p_i}$'s or $\mathtt{x_{ij}}$'s, and $\mathtt{x_k}$'s must be constants, in which case $\mathtt{q}(\mathtt{x_1}, ..., \mathtt{x_a})$ is called a *fact*. For the rest of the paper, "rule" refers only to the case where $\mathtt{h} \geq 1$, in which case each $\mathtt{p_i}(\mathtt{x_{i1}}, ..., \mathtt{x_{ia_i}})$ is called a *hypothesis* of the rule, and $\mathtt{q}(\mathtt{x_1}, ..., \mathtt{x_a})$ is called the *conclusion* of the rule.

The meaning of a set of rules and a set of facts is the smallest set of facts that contains all the given facts and all the facts that can be inferred, directly or indirectly, using the rules. Note that variables occurring in exactly one hypothesis and not in the conclusion of a rule are equivalent to wildcards; their names do not affect the meaning of the rule and can be replaced with _.

Datalog is a database query language based on the logic programming paradigm [8, 2]. Recursion in Datalog allows queries that are not expressible in relational algebra or relational calculus but are essential for querying graph structures of unbounded size.

We consider Datalog with limited extensions—stratified negation, unsafe rules, and additional constraints—for capturing complex graph queries, including extended regular expression based queries. Stratified negation allows negated hypotheses, but they may not appear in cycles in recursive rules; it has much simpler meanings and more efficient implementations than arbitrary negation, by allowing all facts in a relation to be inferred before its negation is needed. Unsafe rules contain variables in the conclusion that are unbound, i.e., that do not appear in any hypothesis; such variables may be left in the arguments of inferred facts and be universally quantified. Additional constraints involve primitive arithmetic, comparison, and Boolean operations on variables that appear in the hypotheses or the conclusion of a rule; they are additional conditions on the values of those variables.

Transforming basic queries and extended regular expression based queries. Complex graph queries can be transformed into Datalog with stratified negation, unsafe rules, and additional constraints. Queries where variables, wildcard, and negation do not appear in constructors of edge labels are transformed as described below. Other queries can be transformed in the same way after each exceptional constructor is first transformed into a distinct new constructor that has an additional argument whose value ranges over possible constructors.

Each constructor $\mathtt{c}$ of an edge label corresponds to an *edge relation* $\mathtt{c}$ that relates source and target vertices and the arguments of the label. An edge from $\mathtt{v_1}$ to $\mathtt{v_2}$ with label $\mathtt{c}(\mathtt{a_1}, ..., \mathtt{a_k})$ corresponds to a fact $\mathtt{c}(\mathtt{v_1}, \mathtt{v_2}, \mathtt{a_1}, ..., \mathtt{a_k})$.

Operations in path-properties expressions correspond to rules that combine relations that capture sub-expressions into relations that capture larger expressions. Edge relations capture the smallest expressions. New relations are introduced to capture larger expressions; the arguments of a new relation are determined as described below. Finally, a special relation is introduced to capture the entire query; it projects the relation that captures the outermost path-properties expression onto the query variables.

Arguments of a new relation include all variables in the expression it captures that also appear in the rest of the query or, if the expression is a repetition or is inside a repetition, all variables except those that are local to the repeated expression and appear only in the expression captured; this takes care of variable scoping, and the requirement of appearance in the rest of the query avoids propagation of unneeded values. In particular, for an expression that represents a path segment, the starting vertex and ending vertex of the path segment are included as the first two arguments of the corresponding relation. For an expression that represents a path segment and whose starting or ending vertex is a wildcard or is not indicated explicitly, we introduce a fresh variable for such a vertex. The fresh variable is effectively a wildcard, so the semantics is preserved. When combining relations that capture smaller expressions into relations that capture larger expressions, shared variables are used to capture equality between the ending vertex of one path segment and the starting vertex of the next path segment.

Wildcards for, and negations applied to, vertices, arguments, and labels are transformed as follows. Wildcards for vertices and arguments are handled as described above by introducing fresh variables. All wildcards for labels are transformed into a special edge relation, $\texttt{anylabel}(\mathtt{v}_1, \mathtt{v}_2)$ for source vertex $\mathtt{v}_1$ and target vertex $\mathtt{v}_2$, and a set of rules of the following form, one for each edge relation `c`:

$$\mathtt{c}(\mathtt{v}_1, \mathtt{v}_2, \mathtt{a}_1, ..., \mathtt{a}_\mathtt{k}) \rightarrow \texttt{anylabel}(\mathtt{v}_1, \mathtt{v}_2) \tag{7}$$

Negation applied to a vertex or an argument is transformed into an inequality constraint attached to the relation that captures the enclosing path-properties expression and where the vertex or argument with negation is replaced by a fresh variable; the inequality constraint expresses that the fresh variable is not equal to the constant or variable to which the negation is applied, except that the constraint is omitted if the negation is applied to a variable not used in the rest of the query. Negation applied to a label is transformed into `anylabel` plus a negated hypothesis, where the hypothesis corresponds to the edge relation for the label without negation.

Combinations of paths using conjunction, disjunction, and negation are transformed as follows. Suppose $\mathtt{p}_1(\mathtt{x}_{11}, ..., \mathtt{x}_{1\mathtt{k}_1})$ captures $\mathtt{exp}_1$, and $\mathtt{p}_2(\mathtt{x}_{21}, ..., \mathtt{x}_{2\mathtt{k}_2})$ captures $\mathtt{exp}_2$. If $\mathtt{p}(\mathtt{x}_1, ..., \mathtt{x}_\mathtt{k})$ captures $\mathtt{exp}_1 \wedge \mathtt{exp}_2$, then we introduce a rule:

$$\mathtt{p}_1(\mathtt{x}_{11}, ..., \mathtt{x}_{1\mathtt{k}_1}) \ \wedge \ \mathtt{p}_2(\mathtt{x}_{21}, ..., \mathtt{x}_{2\mathtt{k}_2}) \rightarrow \mathtt{p}(\mathtt{x}_1, ..., \mathtt{x}_\mathtt{k}) \tag{8}$$

Note that if $\mathtt{exp}_1$ and $\mathtt{exp}_2$ are consecutive path segments, then $\mathtt{x}_{12}$ and $\mathtt{x}_{21}$ are the same variable. If $\mathtt{p}(\mathtt{x}_1, ..., \mathtt{x}_\mathtt{k})$ captures $\mathtt{exp}_1 \vee \mathtt{exp}_2$, then we introduce two rules:

$$\begin{array}{c} \mathtt{p}_1(\mathtt{x}_{11}, ..., \mathtt{x}_{1\mathtt{k}_1}) \rightarrow \mathtt{p}(\mathtt{x}_1, ..., \mathtt{x}_\mathtt{k}) \\ \mathtt{p}_2(\mathtt{x}_{21}, ..., \mathtt{x}_{2\mathtt{k}_2}) \rightarrow \mathtt{p}(\mathtt{x}_1, ..., \mathtt{x}_\mathtt{k}) \end{array} \tag{9}$$

These rules may be unsafe, because any variable in $\mathtt{p(x_1, ..., x_k)}$ that is not in a disjunct is unbound in the conclusion of the corresponding rule. More generally, a conjunction with k conjuncts is transformed into a rule with k hypotheses, and a disjunction with k disjuncts is transformed into k rules. Negation applied to a path-properties expression is simply transformed into a rule with a negated hypothesis; we show in the next section that these negations are stratified.

A repetition of an expression is transformed into a fact with variable arguments, for repeating zero times, and a rule involving recursion, for repeating non-zero times. If $\mathtt{p_1(x_1, x_2, x_3, ..., x_k)}$ captures `exp`, and $\mathtt{p(x_1, x_2, x_3, ..., x_k)}$ captures $\mathtt{exp}^*$, then the fact is $\mathtt{p(x, x, x_3, ..., x_k)}$, and the rule is

$$\mathtt{p(x_1, x_{12}, x_3, ..., x_k)} \ \wedge \ \mathtt{p_1(x_{12}, x_2, x_3, ..., x_k)} \rightarrow \mathtt{p(x_1, x_2, x_3, ..., x_k)} \tag{10}$$

The rule may also be written by exchanging `p` and $\mathtt{p_1}$ in the hypotheses. They are both correct rules, but depending on the query, may lead to different asymptotic running times, as discussed below.

Constraints themselves do not require transformation. If all variables in a constraint are from the same scope after unnecessary scopings are removed, i.e., the constraint is not inside a repetition and involves both local variables and non-local variables of the repeated expression, then it is simply added as a hypothesis of the rule that combines all the subexpressions it constrains. Otherwise, the facts and rules for the repetition are rewritten when combining the repetition with the expression on the left or right that is constrained. For example, suppose (10) has, as an additional condition, a constraint $\mathtt{c(..., y)}$ that is transformed from the same constraint in `exp`, where `y` is a variable not local to `exp`; $\mathtt{q(x_1, x_2, y)}$ captures the expression to the left of the repetition; and $\mathtt{r(x_1, x_2, x_3, ..., x_k, y)}$ captures the combined expression. Then, the fact $\mathtt{p(x, x, x_3, ..., x_k)}$ and the rule (10) are rewritten into

$$\begin{array}{l} \mathtt{q(x_1, x_2, y)} \rightarrow \mathtt{r(x_1, x_2, x_3, ..., x_k, y)} \\ \mathtt{r(x_1, x_{12}, x_3, ..., x_k, y)} \ \wedge \ \mathtt{p_1(x_{12}, x_2, x_3, ..., x_k)} \rightarrow \mathtt{r(x_1, x_2, x_3, ..., x_k, y)} \end{array}$$

Combining a repetition with the expression on the right that is constrained is similar. However, if a constraint involves non-local variables on both sides of the repetition, only one side can be combined using the rewrite above, and the non-local variables on the other side are not bound and must be enumerated during the execution of the resulting program. Therefore, one may choose to combine with the side that will minimize the enumeration.

The transformation into a set of facts and rules has a worst-case time complexity of $O(qvs)$, where q is the size of the query, v is the number of variables, and s is the maximum number of scopes that the variables in a constraint are in. This is because, if all variables in each constraint are from the same scope, then the transformation is linear in qv, since the transformation considers each construct in the query once, and in the worst case, each variable may be an argument in all the intermediate relations. Otherwise, to combine the repetition of each nested scope with an expression to the left or right of the repetition, we rewrite the fact and the rules for the repetition, which yields a factor of s in the complexity. In addition, if a constraint involves non-local variables on both sides of the repetition, and one chooses to combine with the side that minimizes the enumeration during execution, then trying all possible combinations to find the minimum will incur a factor exponential in s.

Handling methods and object creation. The definition of a method is transformed into a set of rules for the path-properties expression in the method body, as described above, except that the relation that captures the entire method is identified by the fully qualified method name, and is related to the class where the method is defined. That relation relates the arguments of the method, including `this`, to the return value, captured by the query variables of the method body. A method invocation is transformed into rules that conclude the relation corresponding to the invocation if the object on which the method is invoked is an instance of a class that defines the method or inherits the method from a class that defines it, and if the relation corresponding to the method holds for the arguments and return value of the invocation.

Objects are created from query results after query evaluation, so object creation does not need to be transformed into Datalog. While multiple logical object ids can be given to an object, for efficient search and equality comparison in subsequent queries and other processing, an object must have a unique physical id for indexing. To achieve this, whenever a new object is to be created and to which search and equality comparison might be applied, it is matched against existing objects, and if found, a reference to the existing object is used, as opposed to creating an identical copy of the existing object.

Example. For example, the query (3) is transformed into a fact $\mathtt{notdefs}(\mathtt{x}_1,\mathtt{x}_1,\mathtt{x})$ and the following rules, where the query result is captured by the relation `result`:

$$\begin{array}{l} \neg\mathtt{def}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}) \rightarrow \mathtt{notdef}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}) \\ \mathtt{notdefs}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}) \ \wedge\ \mathtt{notdef}(\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}) \rightarrow \mathtt{notdefs}(\mathtt{x}_1,\mathtt{x}_3,\mathtt{x}) \\ \mathtt{notdefs}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}) \ \wedge\ \mathtt{use}(\mathtt{x}_2,\mathtt{x}_3,\mathtt{x}) \rightarrow \mathtt{notdefsuse}(\mathtt{x}_1,\mathtt{x}_3,\mathtt{x}) \\ \mathtt{notdefsuse}(\mathtt{start},\mathtt{y},\mathtt{x}) \rightarrow \mathtt{result}(\mathtt{y}) \end{array} \tag{11}$$

The query (4) is transformed into the same rules except with $\neg$`def` in (11) replaced by $\neg$`deforuse` and with two additional rules:

$$\begin{array}{l} \mathtt{def}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}) \rightarrow \mathtt{deforuse}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}) \\ \mathtt{use}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}) \rightarrow \mathtt{deforuse}(\mathtt{x}_1,\mathtt{x}_2,\mathtt{x}) \end{array}$$

5 Generating specialized algorithms and complexity formulas

While Datalog programs can be executed in a Prolog system, recursion could cause nontermination or exponential running time, depending on the order of rules, due to failure to remember computations already attempted. Polynomial running time can be ensured by executing Datalog programs in a tabled logic programming system, such as XSB [24], but it could differ asymptotically, such as between linear and quadratic, depending on the order of hypotheses in individual rules. Also, analyzing the running time requires understanding the execution engine, including for XSB its sophisticated tabling mechanism, and is nontrivial even for experts. Additionally, a light-weight program that is specialized to do only the query at hand and can more easily be plugged into other applications is often preferable to a heavy-weight generic execution engine.

We summarize the method described in [18] for generating specialized algorithms and complexities from pure Datalog, and extend it here to handle stratified negation, unsafe rules, and additional constraints.

Generating algorithms and complexities from Datalog. A method for transforming any set of Datalog rules into an efficient, specialized program with time and space complexity guarantees has been studied [18]. The method breaks any given set of rules into rules that have one or two hypotheses and generates an efficient program that, given any set of facts, computes the meaning of the given rules and facts. The generated program embodies (1) an incremental algorithm to consider one fact at a time and (2) a combination of linked and indexed data structures for the sets of facts and indices used by the algorithm. Overall, each combination of instantiations of the hypotheses is considered exactly once and in constant time.

The method also produces formulas for the time and space complexity of the generated program in terms of data size. Let $\#\texttt{p}$ denote the number of facts that actually hold for relation `p`. A rule with one hypothesis about relation `p` is fired at most $\#\texttt{p}$ times; a rule with two hypotheses about relations $\texttt{p}_1$ and $\texttt{p}_2$ is fired at most

$$\texttt{min}(\#\texttt{p}_1 \times \#\texttt{p}_2.\texttt{matched}, \#\texttt{p}_2 \times \#\texttt{p}_1.\texttt{matched}) \qquad (12)$$

times, where $\#\texttt{p}_2.\texttt{matched}$ denotes the maximum number of combinations of values of arguments of $\texttt{p}_2$ that are not shared with $\texttt{p}_1$ for each combination of values of arguments of $\texttt{p}_2$ that are shared with $\texttt{p}_1$, and vice versa for $\#\texttt{p}_1.\texttt{matched}$; if this number is not known from application domain knowledge, it is bounded by the product of the domain sizes of unshared arguments as well as by the size of the relation. The overall time complexity is the sum of the number of firing times for all rules.

The method applies to pure Datalog, and has been applied successfully to a number of applications, including regular path queries [19] and parametric regular path queries [17], grammar constraint simplifications [18], program pointer analysis [5], and parts of the ANSI standard for role-based access control [4]. A prototype has also been developed to support the applications and experiments.

Handling extensions. We extend the method above to handle variables as arguments in facts, additional constraints, and negations, as follows.

Unsafe rules and transformation of repetitions produce facts that contain variables as arguments. These variables are universally quantified, but we want to avoid enumerating all possible values of them. So we constrain these variables using equality during matching as much as possible, and leave them in the facts when they are not constrained. The complexity calculation is not affected by this optimization since the formulas are for the worst case.

For each constraint attached as an additional hypothesis in a rule where variables in the constraint are bound in other hypotheses, the constraint can simply be evaluated after all its variables are bound with definite values; constraints involving primitive arithmetic, comparison, and Boolean operations can be evaluated in constant time, so such a constraint does not contribute to the complexity formula. For any constraint that contains variables not bound in the other hypotheses, the domains of those variables are numerated. This increases the complexity by a factor linear in the size of the domain for each such variable, but this number will be minimized by the rewrite described in Section 4 for transforming constraints.

Negation applied to a vertex or argument produces an inequality constraint, which is handled as above. Negation applied to a label is transformed into `anylabel` plus a negation applied to an edge relation; the edge relation gives rise to a set of facts, so the negation, i.e., set complement, is easy to compute. Negation applied to an expression is transformed into a rule with a negated hypothesis, so we need to handle Datalog with negation, as described below.

Handling negation. We first show that negations in programs transformed from extended regular expression based queries are always stratified. Note that recursive definitions are only transformed from repetitions. For each relation that captures a repetition, the recursive occurrence of the relation in the hypothesis is not negated, even though the path-properties expression being repeated may be negated. Therefore, there is no negation in cycles formed by the dependency of conclusions on hypotheses in recursive rules.

For stratified negation, we generate a program that fully evaluates a relation before firing any rules that use the negation of the relation. For rules with one hypothesis, if the hypothesis is negative, the program enumerates all values of arguments of the corresponding relation excluding values for which the relation holds; the number of firings is changed from #`p` for a hypothesis about `p` to the product of the domain sizes of all arguments of `p`. For rules with two hypotheses, rather than considering only elements in a relation and elements that actually matched (corresponding to #`p` and #`p.matched`, respectively, in (12)), for a negated hypothesis, we instead consider all arguments of the relation and all unshared arguments, respectively; we pick the order of considering the two hypotheses to give the minimum of two products in a revised form of (12): the number of firings for each rule is the same as before except with relation size #`p` and matched size #`p.matched` replaced by the product of the domain sizes of all arguments and the product of the domain sizes of unshared arguments, respectively.

Example. For example, for the rules and fact in (11) for query (3), the time complexity is $\texttt{O}(\#\texttt{point}^3 \times \#\texttt{var})$, where #`point` is the domain size of the first two arguments of `def` and `use`, and #`var` is the domain size of the third argument. It is obtained from the following sum, one summand for each rule:

$$\#\texttt{point}^2 \times \#\texttt{var} + \#\texttt{point}^3 \times \#\texttt{var} + \#\texttt{use} \times \#\texttt{point} + \#\texttt{point} \times \#\texttt{var}$$

For the rules and fact for query (4), the time complexity formula is the same, except with two additional summands, $\#\texttt{def}$ and $\#\texttt{use}$, for the two additional rules.

Additional optimizations and extensions are possible. The most important optimizations include on-demand, i.e., top-down, computation. In our prototype, we first apply magic set transformations [6] to the rules and the relation that captures the entire query, obtained from the previous section; we then implement the transformed program as described in this section. For query (3), the time complexity after the optimizations is $\texttt{O}((\#\texttt{def} + \#\texttt{use}) \times \#\texttt{var})$. Details of the complexity analysis for on-demand computations using magic set transformations will be presented in a separate paper. We are currently experimenting with the prototype. Handling non-stratified negation is a subject for future study.

6 Related work and conclusion

A number of early studies relate graph analysis problems with regular expressions or regular-expression-like patterns. For example, Tarjan [28] showed that regular expressions provide a general approach for path analysis problems, and he gave efficient algorithms for constructing regular-expression patterns for several kinds of path problems [27]. Regular-expression-like patterns have also been used for static program analysis (e.g., [21]), traversing object graphs in developing adaptive software (e.g., [22, 16]), etc. Most of these works study specific domain problems, and none of them provides a generic and efficient framework for querying complex interrelated objects.

The idea of paths has played an essential role in querying object-oriented databases [14] and semi-structured data [1]. Object graphs may be cyclic but previous query languages do not support patterns that can match paths of unbounded length; this avoids nontermination. Query languages based on XPath [29] use some regular-expression-like features that allow path segments to be skipped but not repeated, and the data are treated as trees, not general graphs. Conditional XPath [20] extends XPath to allow path segments to be repeated, and is as expressive as first-order logic when interpreted on ordered trees, but it does not handle general graphs.

Various forms of regular path queries, allowing general regular-expression-like patterns over general graphs, have been proposed for querying databases and semi-structured data [30, 10, 1, 3, 13, 7]. These languages are studied heavily in terms of expressiveness and query containment, but not on improved ease of expressing queries or efficient implementations. The implementations are basically by transforming queries into logic programs and relying on logic programming engines, such as [23], for query evaluation, but such implementations do not provide precise complexity guarantees.

Regular path queries with parameters have been studied specially for program analysis and model checking [11, 17]. Parameters are essential for expressing correlations of information in different parts of the data, and are needed also in querying system logs for intrusion detection [26], and querying objects in general, as shown in this paper. Drape et al. [12] describe how to code parametric queries as extended logic programs. Liu et al. [17] give complete algorithms and data structures for directly and efficiently solving parametric queries with precise complexity analysis. However, these frameworks do not support a rich data model that can naturally model objects in object-oriented systems and XML data.

The language in this paper is built on parametric regular path queries [17] and a rich object model [14], extending the former with vertex ids, variable scoping, methods, etc.,and extending the latter with powerful regular-expression like patterns. It has the same expressiveness as GraphLog [10], but the support of scoping, and textural flexibilities such as query nesting, make it easier to use, either by itself or as part of another query language such as [15]. The implementation is built on a powerful method for generating specialized implementation with precise complexity guarantees. The transformation to Datalog with limited extensions helps both in understanding the semantics and in implementation. We also extend the method in [18] to efficiently handle stratified negation, unsafe rules, and additional constraints.

Other related works include extensions to OCL path expressions [25] and trace-based program analysis that uses parameters to correlate information along paths [9], but they use more sophisticated heavy-weight mechanisms.

Further extensions and improvements to the query framework can be made. A possible extension is to support universal queries, where properties must hold on all paths in the graph and where a variable is bound to the same value on all paths. In terms of implementation, many optimizations can be explored, including on-demand computation, space reuse, and filtering with constraints. We are applying the query framework to existing and new problems in program analysis, model checking, and security policy analysis.

Acknowledgment. The authors thank Prof. Michael Kifer for references to related work in database research, for helpful explanations and discussions, and for their inspiring work on querying object-oriented and deductive databases.

References

1. S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory*, pages 1–18, 1997.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, Reading, Mass., 1995.
3. S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems*, pages 122–133, 1997.
4. American National Standards Institute, Inc. Role-Based Access Control. ANSI INCITS 359-2004. Approved Feb. 3, 2004. http://csrc.nist.gov/rbac.
5. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994.
6. C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(3-4):255–299, 1991.

7. D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83–92, 2003.
8. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.
9. C. Colby and P. Lee. Trace-based program analysis. In *Conference Record of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 195–207, 1996.
10. M. P. Consens and A. O. Mendelzon. Graphlog: a visual formalism for real life recursion. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of database systems*, pages 404–416, 1990.
11. O. de Moor, D. Lacey, and E. V. Wyk. Universal regular path queries. *Higher-Order and Symbolic Computation*, 16(1-2), 2003.
12. S. Drape, O. de Moor, and G. Sittampalam. Transforming the .NET intermediate language using path logic programming. In *Proceedings of the 4th International Conference on Principles and Practice of Declarative Programming*, Oct. 2002.
13. D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of database systems*, pages 139–148, 1998.
14. M. Kifer, W. Kim, and Y. Sagiv. Querying object oriented databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–402, June 1992.
15. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, May 1995.
16. K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, Mar. 2004.
17. Y. A. Liu, T. Rothamel, F. Yu, S. Stoller, and N. Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 219–230, Washington, DC, June 2004.
18. Y. A. Liu and S. D. Stoller. From Datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 172–183, Aug. 2003.
19. Y. A. Liu and F. Yu. Solving regular path queries. In *Proceedings of the 6th International Conference on Mathematics of Program Construction*, volume 2386 of *LNCS*, pages 195–208. Springer-Verlag, Berlin, 2002.
20. M. Marx. Conditional XPath, the first order complete XPath dialect. In *Proceedings of the ACM 2004 Symposium on Principles of Database Systems*, 2004.
21. K. Olender and L. Osterweil. Cesar: a static sequencing constraint analyzer. In *Proceedings of the ACM SIGSOFT '89 3rd Symposium on software Testing, Analysis, and Verification*, pages 66–74, 1989.
22. J. Palsberg, C. Xiao, and K. Lieberherr. Efficient implementation of adaptive software. *ACM Trans. Program. Lang. Syst.*, 17(2):264–292, Mar. 1995.
23. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Coral—control, relations and logic. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 238–250, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.
24. K. Sagonas, T. Swift, and D. S. Warren. XSB as a deductive database. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1994.
25. A. Schürr. Adding graph transformation concepts to UML's constraint language OCL. In H. Ehrig, C. Ermel, and J. Padberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier, 2001.
26. R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the USENIX Security Symposium*, pages 63–78, 1999.
27. R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, July 1981.
28. R. E. Tarjan. A unified approach to path problems. *J. ACM*, 28(3):577–593, July 1981.
29. The World Wide Web Consortium. XML Path Language (XPath). http://www.w3.org/TR/xpath.
30. M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 230–242, 1990.